Integrated Language Design and Implementation Process

Pacific Software Research Center
March 28, 2000

CONTRACT NO. F19628-96-C-0161
CDRL SEQUENCE NO. [CDRL 0002.28]

Prepared for:
USAF
Electronic Systems Center/AVK

Prepared for:
Pacific Software Research Center
Oregon Graduate Institute of Science and Technology
PO Box 91000
Portland, OR 97291

Software Design Automation: Language Design in the Context of Domain Engineering[1]

Tanya Widen

Fraunhofer Institute for
Experimental Software Engineering
Sauerwiesen 6
D-67661 Kaiserslautern, Germany
widen@iese.fhg.de

Dr. James Hook

Oregon Graduate Institute of
Science and Technology
P.O. Box 91000
Portland, OR 97291-1000 USA
hook@cse.ogi.edu

Abstract

Domain-specific languages improve the productivity of application engineers by raising the level at which they define domain instances. However, more support for designing and implementing these languages in practice is needed.

The Software Design Automation (SDA) method is a specific approach for doing domain-specific language design and implementation based on a principled, semantics-based approach to language definition and implementation.

This paper articulates SDA as a software development method to be used in the context of existing domain engineering methods.

1. Introduction

Throughout the history of computing, domain-specific languages (DSLs) have codified knowledge and increased the productivity of software developers. Significant DSLs include the Formula-Translator (FORTRAN), yet another compiler compiler (yacc), spread sheet languages, and hyper-text markup languages. Traditionally, DSLs have been created opportunistically by visionary experts. Also, techniques for designing and implementing these languages have been inadequate.

Domain engineering provides an opportunity to systematically look for opportunities to introduce DSLs as a reuse mechanism. Domain engineering methods support the study of families of software systems. They provide mechanisms for determining the commonalities and variabilities of a domain, and for creating some reusable assets within a domain. DSLs can also be created within this context to encapsulate reusable information.

The Software Design Automation (SDA) method augments existing domain engineering methods with a principled, semantics-based approach to language definition and implementation. SDA integrates concepts from formal semantics, functional programming, and type theory to produce domain-specific languages with provably correct implementations, and statically checkable global properties.

Thus, domain engineering provides opportunities for creating DSLs and SDA provides support. Throughout an application of the SDA method, a team of method experts work together with the team of domain experts involved in the domain engineering process.

Implicit in the formation of this process is the as yet unproven hypothesis that principled language design can be practiced as a reuse principle by software developers; it is not the sole province of highly trained experts. Articulating a process is the first step in testing this hypothesis.

This paper describes the SDA method. The method has evolved from PacSoft's previous work on the Software Design for Reliability and Reuse (SDRR) project [4,7,27].

An example is used to illustrate important products of the method. Throughout the paper the example text is separated from the method description in boxes to distinguish parts of the method from parts of the illustration.

2. The Example

The example used throughout this paper is a subsystem of a Command, Control, Communications, and Intelligence (C^3I) system. A C^3I system receives, analyzes, and broadcasts messages to and from other systems such as sensor arrays, databases, and human operators. One critical subsystem is Message Translation and Validation (MTV), which validates incoming messages and then stores or rebroadcasts messages in different formats. The MTV subsystem is structured as a collection of modules, each of which is responsible for defining a data structure that can represent the sensor information for a particular message format, performing simple data validation tests, and providing parsing and unparsing functionality for the various messages' representations.

When building MTV systems, application engineers are presented with a semi-formal message specification (a message format) and asked to build an Ada module with the

1. The research reported here was supported by contracts with the USAF Materiel Command. Contracts F19628-93-C-0069, F19628-96-C-0161

appropriate functionality. The problem is automating the task of building a module from a specification. The solution described here is the Message Specification Language (MSL) [28] developed as part of the Software Design for Reliability and Reuse (SDRR) project [4].

3. Software Design Automation

The SDA method describes how to design and implement DSLs within a domain engineering project. The method consists of three phases: analyze the domain, define the language, and implement the generator. A high level view of the method is pictured in Figure 1.

SDA assumes that the domain engineering method being used supports selecting the domain and eliciting and gathering necessary information. This information is captured in a domain analysis document that is taken as input by the SDA process.

In the analysis phase domain information from the domain analysis and domain experts is formalized in models that are used for the language design. In the language definition phase the complete specification of the language is defined based on the models collected in the first phase. This definition is captured as an interpreter expressed in a functional language, which is a prototype of the language that can be used and validated. In the implementation phase a generator is built from the validated language definition. Additional support products are also developed.

1. **Analyze the Domain**
   - 1.1. Capture a written definition of the domain
   - 1.2. Develop model
     - *1.2.1. Model the domain*
     - *1.2.2. Model the solution*
     - *1.2.3. Model the run-time environment*
   - 1.3. Preliminarily validate models
2. **Design the Language**
   - 2.1. Define the language
   - 2.2. Formalize the semantics
   - 2.3. Validate language design
3. **Implement**

Figure 1. High-level View of the SDA Process

The conceptual architecture of a component generator produced by the SDA method is given in Figure 2. The DSL is the user-visible language for specifying components. The domain and solution models are representations in a functional language of abstract models of the problem-view of the domain and the functional behavior of solutions, respectively. The concrete component is a code component that meets the requirements of the run-time environment.

The simple domain semantics map the DSL, which is only required to be expressive over distinguishing characteristics of domain instances, on to the complete domain model. The solver maps the problem level domain model on to the lower level solution model. The emit function is not required by all domains; it translates the implementation independent functional specification of the solution into a concrete artifact in the language technology of the target environment.

The simple domain semantics and solver functions are initially prototyped in SDA as semantics-based interpreters in a functional language. They may subsequently be refined into more sophisticated translators.

$$\begin{tikzcd}
\text{Domain-Specific Language} \arrow[d, "\text{simple domain semantics}"] \\
\text{Domain Model} \arrow[d, "\text{solver}"] \\
\text{Solution Model} \arrow[d, "\text{emitter}"] \\
\text{Concrete Component}
\end{tikzcd}$$

Figure 2. Conceptual Architecture of SDA generator

3.1. Analyze the Domain

The analysis phase determines the requirements of the DSL. Information from the domain analysis and domain experts is first captured in an informal written domain definition and then extended, formalized and validated in three models which characterize the domain, the solution, and the run-time environment. These models provide the basis for the language design.

The domain model will characterize problem instances, the solution model will functionally characterize a problem solution, and the environment model will constrain that solution to reflect the realities of the technology being developed.

Ultimately, the DSL being developed will be a declarative language for describing problem instances and the DSL's implementation will be a generator that calculates instances of the solution, satisfying the constraints of the environment, from problem descriptions in the DSL.

3.1.1. Capture a Written Definition of the Domain

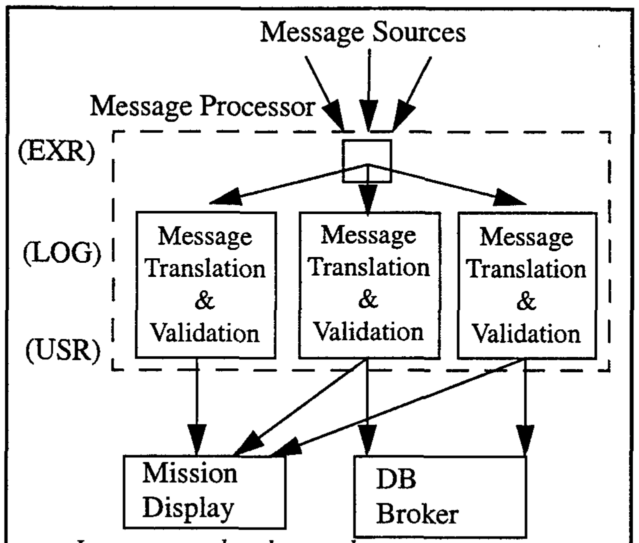
The information collected in the domain definition provides the SDA method experts with an overview of the domain. It also captures information that supports the language design process.

In this step information is gathered from the provided domain analysis documents, as well as from the domain experts and other sources as needed. The information gathered is captured in four parts:

1. Problem statement
2. Architecture
3. Initial requirements
4. Workflow analysis

The problem statement explains the purpose of software components in the domain, enumerates the basic concepts in the domain, and summarizes the external functions that domain instances provide. The example problem statement is captured in the introduction to the example domain in Section 2.

A system architecture captures the surrounding context and scope of the domain. It identifies the neighboring domains that interact with the current domain and it defines the boundaries and interfaces of the domain. Figure 3 illustrates the basic architecture of the MTV domain.



Message Sources
Message Processor
(EXR)
(LOG)
(USR)
Message Translation & Validation
Message Translation & Validation
Message Translation & Validation
Mission Display
DB Broker

In a command and control system a message processor accepts incoming messages, determines the message formats, and dispatches them to the appropriate MTV component. The output of the MTV component is then passed on to other components by the message processor.

Figure 3. MTV Architecture: Message Processor

Initial requirements describe the possible inputs and outputs of the domain as well as the features or actions domain instances may offer. The requirements specification also describes behavioral constraints on domain instances. The initial requirements for the MTV domain are briefly summarized in Figure 4.

The high level functional requirement of MTV is that components process input messages. Processing may include validation of messages received and translation to other formats.

Figure 4. MTV Initial Requirements

The workflow analysis captures information about how domain artifacts are currently produced. This includes information on the processes used, the communication patterns of domain experts as well as the information shared, the assets (including tools) used and produced, and the people involved.

The workflow analysis provides valuable information on the existing notations, or specification language, used by the engineers. It also supports language design through identifying people and products for obtaining additional information, and feedback. The workflow analysis for the MTV domain is summarized in Figure 5.

Informal message specification notation was already in use. Each message format was described by an Interface Control Document (ICD). ICDs are field by field descriptions of a message presented in tabular form. Each field is numbered, the range of expected values is described, any delimiters that are expected are described, and comments on validity and units of the measurement are associated with each field in natural language.

Figure 5. MTV Workflow Analysis

3.1.2. Model the Domain

The domain model structures the common and varying domain requirements from a problem-oriented view of the domain.

Three activities contribute to the formation of the domain model:

1. Identification / abstraction of domain concepts
2. Searching for mathematical abstractions
3. Formalization of the model

Identification and abstraction of domain concepts involves eliciting, describing, and modeling the common and varying concepts of the domain. The entities in the domain are captured as compositions of their parts. The relationships among concepts are also captured. These are integrated into the composition of the entities. Common and varying operations of the domain are also modeled. Concepts from the example domain are illustrated in Figure 6.

The basic building blocks for messages are fields. Atomic fields include representations of integers, characters, strings, and enumeration types. Some fields are delimited.

In addition to fields, there are constraints on messages. These are simple first-order predicates on the contents of the fields in a message based on simple arithmetic formulae, equalities, and inequalities.

Figure 6. MTV Concepts

Typically, the domain model has many levels. The levels are explored in both a top down and bottom up fashion depending on the understanding of the concepts being modeled. The higher levels are refined and the low levels are abstracted as domain understanding grows, until there is a complete model that spans the desired levels.

Known abstractions, which can be applied to the current domain, are actively sought during the analysis. These provide insight and can be reused in the current domain. For example, the solutions or certain properties of the abstractions may be reused in the domain of study.

Of particular interest are mathematical models. Examples of these are: graphs, grammars, or finite automata. Such problems have been well studied and this knowledge may be transferable to the domain being studied. For example, the mathematical models may provide additional properties that can be rigorously analyzed. The mathematical model for the example domain is captured in Figure 7.

The variation in this domain derives exclusively from the different message formats that are to be processed and validated. A message format description describes a set of messages in an external format. Thus, the user view of the domain is the "language" of messages to be processed.

The variation in the MTV domain is naturally modeled by a grammar describing the syntactic structure of the language. (In expressive power the languages appear to be regular, however neither regular expressions nor left- or right-linear grammars appear to be a good match conceptually.)

Figure 7. MTV Mathematical Model

Once a mathematical model has been selected, SDA advocates building an executable representation of the model in a typed functional language, such as Haskell. It is then illuminating to build instances corresponding to particular examples in the domain in the functional language. This step both tests the adequacy of the model and provides a foundation for subsequent validation and exploratory development at later stages of the method.

Getting the model right is the most difficult and critical step of the process. Executable representations in typed languages prove useful because: (1) naive models often contain type errors, (2) type checking enforces all structural constraints on model entities, providing assurance that the examples really are covered by the model, (3) the process of developing examples tends to illuminate regularities in the domain that may lead to more abstract models, and (4) the executable model representation will be the basis of subsequent prototype development.

The final model should be precise, simple, adequate, relevant, and consistent [5]. A precise model is externally correct or unambiguous. In a simple model, domain concepts are structured so that they are easily understood. An adequate model conveys all of the information needed. A relevant model contains only the information needed and nothing else. Finally, a consistent model contains no conflicting information.

3.1.3. Model the Solution

The solution model captures the solution view of the domain concepts. This model also has to distinguish the common and varying requirements of the domain, but at a lower level than the domain model. The solution model refines the system architecture of the domain, expressing the generic architecture of generated components. The solution model distinguishes between common and varying aspects of the design of the domain so that the design can be suited to the individual instances.

The solution model divides the domain into the modules of related functionality. These may be required for all domain instances, or may depend on the particular instance. The interfaces to the modules, sometimes known as their signatures, are captured in a functional language as well as in the target language of the domain if necessary. The target language is the language to be generated during domain instance creation.

On top of the module signatures, the connections, or interactions between modules are captured. These indicate the flow between modules. Constraints on the solution are also captured, as well as configurations of the modules. These provide additional information on how the modules can be combined, or how the modules interact. The solution model for the example domain is captured in Figure 8.

The solution model can be captured as an Ada Package that includes a declaration of the datatype providing the internal representation, a parser and unparser for the external representation, a parser and unparser for the character representation, and validation functions that may be applied to messages in the external or character representations. In the domain analysis this package specification was presented in Ada PDL (Program Description Language).

Figure 8. MTV Solution Model

3.1.4. Model the Run-time Environment

In addition to the architecture of the solution, it is necessary to characterize the environment in which components execute. This somewhat ad hoc collection of requirements is called the environment model. It includes (1) the implementation technology of the target system, (2) significant reusable assets (libraries) provided in the target environment, and (3) non-functional constraints on the behavior of the target system (e.g. performance, stack and heap usage).

The legacy environment for MTV specified that Ada was the target language and included a minimum performance requirement.

Figure 9. MTV Environment Specification

The requirements of the environment influence the technology that may be used to implement generation. In the initial phase of the SDA project, tool support was developed supporting source level integration of components (this component is the emitter in Figure 2). That is, PacSoft developed a retargetable compiler tool that produces Ada or C that can be linked directly with existing legacy code [22]. In ongoing work, PacSoft is developing tool support for "object-level linking" using standard interfaces expressed in COM or CORBA. The example environment specification is captured in Figure 9.

3.1.5. Preliminarily Validate Models

Having assembled the three models: domain, solution and environment, it is now time to mentally kick the tires and make sure the models are sufficiently coherent and at an appropriate level before investing in the subsequent development of a formal language description. This step is called validation. The application of common sense in validation is critical!

Specific activities that build confidence in coherence include: (1) doing a preliminary design review in which it is argued informally that the solver (see Figure 2) can be implemented (that is, show that all variable components of the solution are uniquely specified by the domain model), (2) prototyping combinators that may be used to glue aspects of the solution together (ideally these combinators will be well behaved algebraically, frequently they will expressible as a monad), and (3) reviewing worked out examples with domain experts.

A summary of the validation of the example domain is captured in Figure 10.

The interpretation of the model was explored by prototyping it in ML. The prototype focused on the parsing problem, as it appeared the most challenging. A monadic combinator based parser was developed. It used ML types to represent the data structure and the structure of the ML program to implicitly represent the grammar. Basic parsers for the atomic field types were implemented as ML functions. These were combined systematically with monadic combinators to implement concatenation, sequential product construction and alternation.

This style of prototyping made extensive use of ML's typed functional features: higher-order functions, Hindley-Milner type inference, and algebraic datatype declarations. It could have equivalently been developed in Haskell.

Figure 10. MTV Validation Summary

3.2. Design the Language

After decades of research, language design remains one of the most difficult and challenging activities in computer science. SDA advocates a particular approach to language design that produces typeful, formally defined languages. However, language design remains an art requiring diverse skills, experience, and judgement.

In SDA, the language design problem is essentially this: define a language in which problem instances can be effectively specified by domain experts and from which solutions may be mechanically calculated. In addition, the language should be principled, that is it should incorporate the language design principles articulated by Tennent [20]. It should be typed, support modularity and reuse, have internal regularity, and, as suggested by Einstein, be as simple as possible (but no simpler).

The method is designed to inform language design with as much relevant information as possible. The existing communication patterns of the experts may already contain the kernel of an effective notation. The formal definition and validation of the domain model and solution model suggest an appropriate target expressive power and provide prototype mechanisms for assembling the semantic components.

While the studies and models prescribed by the method are informative, they are not a substitute for user (or domain expert) feedback. It is critical that a "real user" be available to the language designer for informal interaction and that feedback from a community of users be systematically solicited periodically.

In addition to the general principles of language design, SDA recommends the following principles for DSL design:

- Focus on a declarative description of the problem, not an imperative description of the solution. Language design driven by people with intimate knowledge of a particular solution is often overly biased toward that solution. An excellent example of a radically declarative language is NASA's Amphion/NAIF system developed by Lowry and others [8].
- Imitate good languages. Whenever appropriate imitate the lambda calculus, Algol, Pascal, Prolog, Haskell, ML, or other well studied "good" examples.
- Never invent what you can steal. SDA advocates two primary approaches to reuse in language design. The first is to prototype domain-specific languages as embedded languages in higher-order, typed languages such as Haskell or ML. This provides a rich, well-understood type structure, basic mechanisms for abstraction, control, and modularity, and gives a flexible environment for prototyping. The second reuse strategy is to use monadic building blocks to construct the semantics of the language out of reusable pieces (a reasonable alternative would be to use Mosses's action semantics [10]).
- Avoid becoming general purpose by accident. Be aware of the expressive power of the language you are defining. If it can express arbitrary computation make sure it does it well.

The language definition phase produces a preliminary DSL implementation as an embedded language with the simple domain semantics and solver expressed as interpreters in a functional language so that it can be used and evaluated. This phase is divided into three parts:

1. Language definition
2. Semantics formalization.
3. Language validation

3.2.1. Define Language

The DSL will be used to specify domain instances. Only the distinguishing features of domain instances need to be expressible in the language. Generating instances from specifications unites the common and varying features into a complete solution.

The language design proceeds with the design strategy and the type system / gross syntactic structure of the language.

In MSL, there are three kinds of statements corresponding to the three basic concepts.

1. Type declarations derived from the message type structure.

2. Parsing declarations will potentially introduce data for any type declared in the language

3. Constraint declarations will be built around a term language expressive over the types of the language with sufficient expressive power to express first-order logic. In the full language, which includes list and array type constructors, support for quantification over elements of these types is also required.

1. Type Constructors

1. Base types:
- *Integer*
- *Integer subrange*
- *Character string*
- *Character string of bounded length*

2. Simple type constructors:
- *Labeled products (records)*
- *Labeled sums (variants, enumeration types)*

3. Functions: Since the language is first-order no general function type constructor is included in the type language. There is a notion of function, and functions are typed, but since functions are not first-class values the function type is not included in the language of types.

2. Relationships between types

***Type Equality.** Two types are equal if they have identical structure. Two labeled products have identical structure if they have identical label sets and the structure of all corresponding labeled component fields are identical.*

***Type Ordering.** A simple form of subtyping is induced from the integer subranges and string length declarations.*

***Type Abstraction.** Types are named at top level, but there is no notion of polymorphism or quantification over types.*

Figure 11. MSL Language Definition

The design strategy is an informal statement that identifies the key concepts, metaphors, and models to be pursued in the language definition. It identifies the concepts and notations from the models and workflow products that will organize the type system and motivate the gross syntactic structure of the language.

The type system and gross syntactic structure of the language identify the atomic entities, or types, and entity composition operations appropriate for each type so that all composite types can also be created. At a minimum there must be methods in the language to construct entities of all meaningful types. Typically there will also be constructions that analyze entities of each type as well. When both a construction and analysis mechanism are present they should be coherent and satisfy the "correspondence principle" of Prawitz [15]. The language definition for the example domain is captured in Figure 11.

3.2.2. Formalize the Semantics

A formal semantics makes explicit the relationship between the concepts introduced in the type system and syntax of the language and the abstractions identified in the domain model and solution model. To decompose the problem of giving language meaning, SDA recommends that the simple domain semantics and the solver be considered separately, and that the solver be further decomposed into separate solvers for each identified facet of the solution.

The first definition of the DSL to be formalized is the simple domain semantics. This is a (sometimes trivial) interpretation of the DSL into specific instances of the domain model.

The next step is to specify the solver, which maps problem specifications to solutions. Recall that at an earlier stage the feasibility of this translation was asserted informally; this relationship is made explicit now by completing the explicit formal definition of the solver.

The structure of the solution model will be reflected in the structure of the solver. If the solution model can be easily decomposed, that decomposition can be exploited. If algebraic combinators for building solution components were developed earlier, they can be exploited in the compositional construction of solutions. The semantics of MSL is described in Figure 12.

SDA suggests building solvers that are structured as semantics-based interpreters of the domain model. Other technologies may be supported in the future. In a semantics-based approach, underlying abstractions can be used to structure the solution. Many critical facets of the semantics of traditional languages are expressed using a simple categorical concept called a monad (or triple) [9]. This mechanism allows semantic concepts to be expressed and combined in a more modular fashion than has been previously possible. It also provides structure and guidance to the design space of language features [24,25]. In particular, it is possible to articulate a short list of semantic features and characterize their potential meaningful interactions. This added structure simplifies the problem of language design.

SDA supports this style of principled language definition in two ways: a methodology that builds on the rich tradition of semantics and the recent results in monadic abstraction, and a tool kit that allows implementations to achieve the reuse promised by the method.

In the case of MSL, the domain semantics essentially unfolded all the named intermediate values (which were introduced to satisfy the principle of abstraction and subsequently enabled significant reuse) and yielded a single parsing action that matched exactly the message type being specified. The abstract syntax of the source language and the domain model were sufficiently similar that the closed, irreducible terms of the abstract syntax were used to represent instances of the domain model.

The solver for parsing is given by providing appropriate interpretations for the primitive operators and mapping the meaning of composite operators (such as sum and product) onto the corresponding monadic combinators. In this way a parsing semantics can be given to the domain model that is independent of which representation is being parsed [28]. Similarly, an almost identical representation independent unparsing semantics can also be given. Curiously, the unparser does not appear to be abstracted over a monadic structure [28].

Figure 12. MSL Semantics

3.2.3. Validate Language Design

As the language is developed, it is validated internally by the team as well as externally by the intended users. Validation ensures that the language definition is correct and complete. Feedback from the intended users is key for usability aspects of the language. This includes the ease of transitioning the language into practice once it is finished.

If formal validation is required, it is possible to characterize and prove properties about the simple domain semantics and the solver that address critical correctness issues. For example, Walton has proven properties about the MSL generator that characterize the correct interaction of parsing and generation solution modules [27]. Similarly, Peyton Jones, Meijer, and Leijen have been able to prove associativity of a "parallel composition" operation for a domain-specific language for animation behaviors in COM [14].

3.3. Implement

Implementing the generator is the last phase of the SDA method. At this point the language has been defined and a prototype implementation has been developed as an embedded language in a functional host language. It is now necessary to build a tool that can be used by the application engineers.

The first step is to critique the prototype. If it is adequate there is no reason to proceed with further refinement of the implementation. Typically, the prototype may fail to be adequate because: (1) the integration requirements in the environment model require the two-stage form of a compiler or code generator, (2) the interpreter does not meet the constraints of the run-time environment model, or (3) the user interface of the interpreter may not be suitable for the intended user community. These issues are discussed below.

The implementation strategy selected was to implement a traditional compiler architecture to translate from MSL to an ML-like functional language from which we could use other tools to calculate an implementation in Ada. An alternative implementation strategy involving an interpreter was explored by Tolmach [21]. His interpreter reuses the front-end and type-checker of the compiler described below.

Figure 13. MSL Implementation Strategy

Staging

An interpreter is a program that contains an *eval* function that maps a program into a *behavior* function expressed in the same execution environment as *eval*. In contrast, a compiler (or a generator) is a program that contains a *compile* function that maps a program into an executable object that when subsequently executed behaves according to the specification of the program. This difference between interpreters and compilers is characterized by their execution stages. The interpreter has a single stage, containing both *eval* and the *behavior*. The compiler or generator has two stages: "compile-time" when the *compile* function is executed and "run-time" when the specified behavior occurs.

The methods for prototyping the solver as an interpreter produce a single-stage component. Single-stage systems may be appropriate for some environments, particularly ones in which performance is not critical and a high-level interface (such as COM or CORBA) can be exploited for connection with other system assets.

In performance critical systems that require code-level integration in legacy languages (such as C or Ada) a translator is typically required. Sheard, Taha, and Benaissa have developed extensive tools and techniques based on typed-metaprogramming for the systematic conversion of single-stage interpreters into multi-stage translators [19]. These tools are part of the SDA technology.

Environment Constraints

In many applications, the generated solutions must interoperate closely with legacy code expressed in a conventional imperative language, such as C or Ada. In this case a systematic translation of the behavior from the specifications in the functional prototype to the legacy language is required. (This is called the emitter in Figure 2.) PacSoft has developed extensive compilation infrastructure to support this. Building on ideas of Kieburtz and Volpano [23], Oliva and Tolmach developed a highly parameterized functional language compiler called RML (Restricted ML) [22]. RML compiles a simple functional language to C or Ada in a type faithful way, and has extensive mechanisms for encapsulating legacy assets (such as libraries) so that they may be used efficiently by the generated code.

To use RML to implement an emitter, the solver is first staged (either manually as was done with MSL or automatically with the tools described above), then the functional program expressing the behavior of the generated component is compiled to the legacy language by the RML compiler.

Appropriate User Interface and Analyses

The third typical shortcoming of the prototype language environment is the appropriateness of the tool for use by the intended domain experts. Specific issues here can include (1) unintuitive interaction between the host and embedded language (e.g. complex type errors that expose implementation details), (2) opportunities for a more refined (or possibly more liberal) type discipline than that inherited from the host language, (3) the need to produce high-quality error messages, and (4) the opportunity to have a non-textual representation of the language.

Currently all of these "front-end" and analysis issues are addressed in SDA by using conventional compiler implementation technologies and methods. For example, the lexical analysis and parsing in the MSL implementation was developed with ML-lex and ML-yacc. The type inference algorithm was coded by hand in SML.

Another issue to be considered when critiquing the prototype is if the DSL provides opportunities for tools other than the generator that might be useful to the users. These tools might include analysis tools or test-case generators. Often these may be defined in the context of the semantics-based interpreter as a "non-standard semantics" of the model.

Once the critique is complete a design and implementation plan for the language environment must be developed and executed. The implementation strategy and plans for the MTV domain are captured in Figure 13 and Figure 14.

4. Related Work

Most well known domain engineering methods aim at creating libraries of reusable components or domain-specific architectures, not DSLs. Examples include: Model Based Software Engineering, Organizational Domain Modeling, and Synthesis [16,18,17]. These methods typically include activities for selecting/scoping a domain, gathering information on the domain and designing and developing reusable assets for the domain. SDA aims to be integrated with these types of methods. SDA enhances domain engineering methods with activities for designing and developing DSLs for generating reusable components, which can be integrated with the rest of the domain engineering products.

ML versions of standard compiler tools for lexical analysis and parsing were used to construct the front end. Abstract syntax was expressed as a polymorphic ML datatype. The polymorphism allowed for the type system to enforce fine distinctions in the level of analysis and rewriting applied to the abstract syntax.

The simple domain semantics were expressed as a catamorphism (a particular kind of simple structured recursive function definition) over the abstract syntax. The back-end directly reflected the structure of the solver, with the regularity of generation of the parsing and unparsing operations manifest in the structure of the code.

A library implementing the primitive operators and the monadic combinators was provided in the ML-like target language of the DSL-specific compiler described here. In the initial implementation this library was implemented from minimal primitives in the ML-like target language. This system, while reliable, performed poorly due to code bloat [28,13]. Once the abstract machine became stable, this implementation was replaced by an implementation in a hybrid of the ML-like target language and Ada [21]. This reimplementation exploited specific mechanisms for developing such hybrid implementations in the RML compiler [22].

Figure 14. MSL Implementation Plans

Other projects investigating the design and development of DSLs include work on the Draco project, the GenVoca project, the Amphion project, and the FAST method. A brief introduction to these projects and their relation to SDA is described below.

The Draco approach to domain engineering is described through the products that are created and input into the Draco mechanism [11,12]. The Draco mechanism takes a description of a DSL in six parts:

- a parser for the language objects and operations,
- a prettyprinter,
- optimizing transformations,
- reusable components which capture the various semantics for language objects and operations,
- generators for language parts that can be generated automatically to code,
- and analyzers for the language

System descriptions in a DSL are then transformed, with human guidance, into working systems through the Draco mechanism.

This approach differs in a couple aspects from SDA. First, the transformations in Draco require human guidance. Second, the process for designing languages and for creating the products which are input into Draco is not well defined, and neither are the products.

The GenVoca project studies how to build software system generators [3]. Software system generators are tools for assembling software from interchangeable reusable components. Therefore, this is a compositional approach to languages whereas SDA is a transformational approach.

The DSL constructs consist of the ways to capture the possible compositions of components. These are not determined through a language design process, but rather from the architecture of the components.

The Amphion project creates languages to help non-programmers better use software libraries [8]. A domain theory is created for each domain which relates the problem area to the subroutines available in the library. A problem statement is captured as a relation between the inputs and outputs. A theorem prover is used to materialize the relation using deductive synthesis and generate a working program.

The Amphion approach has a similar conceptual architecture to SDA's (see Figure 2). The difference is that the solver uses a deductive theorem prover instead of a semantics based interpreter. Another difference between Amphion and SDA is that there is no documented method on how to create the libraries or the languages, only on how to capture the domain theory which relates the two.

FAST is a domain engineering method for analyzing software families and for developing DSLs for them [2]. FAST includes a well documented and validated commonality analysis subprocess, which groups domain experts together to gather, analyze and capture domain information. Other steps in the FAST process aim to build on this analysis to define and develop a DSL, an application engineering environment, and a standard application engineering process. However, these steps are not well defined in the FAST literature as are the commonality analysis steps.

In summary, the SDA approach is similar to other DSL approaches in that tools and techniques for designing and developing DSLs is studied. However, other approaches do not have defined methods for language design, as this paper attempts to provide for SDA. The other approaches focus more on the implementation of the language.

5. Conclusion

In the past, systematic approaches to domain-specific language design have been inadequate. Domain engineering provides an opportunity for designing and developing DSLs. SDA provides a systematic, tool-supported approach for the design and implementation of domain-specific languages, built on the rich tradition of language principles that are the core of the programming language community.

Over the past five years, PacSoft has studied domain-specific languages. Our emphasis has been on languages for the description of systems components and the automatic generation of these components. We have developed a set of tools and techniques for DSL design and implementation that focus on the application of semantics and programming language principles [4,22]. Using these tools we have developed a reliable DSL for Message Specification and Translation that has been experimentally validated and is incorporated in an Air Force demonstration command and control system [4,27]. We have also worked with Lucent on the design of languages for use in telephony [26].

Explicitly defining the SDA method is a first step in testing whether principled language design can be practiced as a reuse principle by software developers, and that it is not the sole province of highly trained experts. The integration of SDA with domain engineering methods provides a mechanism for bringing these "academic best practices" into engineering use. There, they can be applied and evaluated on real world problems.

We are actively collaborating with domain engineering researchers and practitioners at Lucent Technologies to test our ideas in the real world, and to evolve the process based on these experiences.

Currently, there are still some significant barriers, not the least of which is the educational and conceptual background necessary to execute the SDA method. We continue to address these issues by developing a curriculum that focuses on the design of domain-specific languages as an engineering practice.

Software Engineers need to learn the principles of language design not just because it is part of the core body of knowledge that every computer scientist should know, but because they will need to know it to effectively build domain-specific languages, an important form of reusable assets.

Acknowledgments

The authors would like to thank Walid Taha for his time and suggestions for improving the paper.

References

[1] Arango, G. and Prieto-Diaz, R. (eds.) Domain Analysis and Software Systems Modeling. IEEE Computer Society Press, 1991.

[2] Ardis, M. and Weiss, D. Defining Families: The Commonality Analysis, in Proceedings of ICSE 19, pp. 649-650, May 1997.

[3] Batory, D. et. al. The GenVoca Model of Software Systems Generators, IEEE Software, pp. 89-94, sep 1994

[4] Bell, J. et al. Software design for reliability and reuse: A proof-of-concept demonstration. In TRI-Ada '94 Proceedings, ACM, November 1994.

[5] Draghicescu, M. The role of model building in domain analysis. OGI-CSE-TR, July 1996.

[6] Hook, J. and Walton, L. The Design of Message Specification Language. OGI-CSE-TR, June '97.

[7] Kieburtz, R. et al. A software engineering experiment in software component generation. In ICSE '96 Proceedings, IEEE, 1996.

[8] Lowry, M., Philpot, A. Pressburger, T., and Underwood, I., AMPHION: Automatic Programming for Scientific Subroutine Libraries. ISMIS 1994.

[9] Moggi, E. Notions of Computation and Monads Information and Computation, 93(1), 1991.

[10] Mosses, P. Action Semantics. Cambridge Tracts in Theoretical Computer Science, Number 26, Cambridge University Press, 1992.

[11] Neighbors, J. Draco: A method for engineering reusable software systems. Software Reusability Vol 1, Biggerstaff, T., and Perlis, A., Eds., Addison-Wesley. pp 295-319, 1989.

[12] Neighbors, J. The Draco approach to constructing software from reusable components. IEEE Transactions on Software Engineering, vol 10 pp 564-574 September 1984.

[13] Oliva, D. Baseline performance measurements of un-optimized generated code. OGI-CSE-TR, February 1995.

[14] Peyton Jones, S. et. al. Scripting COM Components in Haskell. To appear in Proceedings of The Fifth International Conference on Software Reuse June 1998.

[15] Prawitz, D. Ideas and Results in Proof Theory. Logic and the Foundations of Mathematics, Vol. 63, pp. 235-307, North-Holland, 1971.

[16] Software Engineering Institute. Model-Based Software Engineering. Web pages, http://www.sei.cmu.edu/technology/mbse/, January 1998

[17] Software Productivity Consortium. Reuse-Driven Software Processes Guidebook. Technical Report SPC-92019-CMC, Version 02.00.03, Nov. 1993

[18] Software Technology for Adaptable Reliable Systems (STARS). Organization Domain Modeling (ODM) Guidebook, Version 2.0. Unisys STARS Technical Report STARS-VC-A025/001/00, Reston VA, June 1996.

[19] Taha, W. and Sheard, T. Multi-Stage Programming with Explicit Annotations, PEPM '97, Amsterdam, June 1997.

[20] Tennent, R. D. Principles of Programming Languages. International Series in Computer Science. Prentice-Hall International, London, 1981.

[21] Tolmach, A. Elaborating the specification of Message Specification Language. OGI-CSE-TR, July 95.

[22] Tolmach, A. and Oliva, D. From ML to Ada. To appear in the Journal of Functional Programming, original version May '97.

[23] Volpano, D. and Kieburtz, R. Software Templates. In the Proceedings of the ICSE 8, pp. 55-61, August 1985.

[24] Wadler, P. Comprehending monads. Mathematical Structures in Computer Science, Special issue of selected papers from 6'th, Conference on Lisp and Functional Programming, 1992.

[25] Wadler, P. Monads for functional programming. Marktoberdorf Summer School on Program Design Calculi, Springer Verlag, NATO ASI Series F: Computer and systems sciences, Volume 118, August 1992.

[26] Walton, L. Domain Analysis for Switch Maintenance Configuration Control. OGI-CSE-TR, June '96

[27] Walton, L. and Hook, J. Creating and verifying domain-specific design languages. OGI-CSE-TR, September 1995.

[28] Walton, L. and Hook, J. Message Specification Language (MSL): A domain-specific design language for message translation and validation. OGI-CSE-TR, September '94.